DECEMBER 1, 1984

PRICE $5.00

/C

THE JOURNAL FOR C USERS

VOLUME 1 • NUMBER 2

The C standards committee

The American National Standards Institute's X3-J11 committee, the C standards committee, has been working on an informal draft of a standard since the beginning of 1984. Although the draft is still fluid (subject to change), this column will cover the significant proposals that achieved informal approval at the September meeting in Boston.

One major subject of discussion at the September meeting involves the preprocessor. In UNIX, the preprocessor is a text filter, converting the text of the C source code into another text file. Substitutions are performed, based on `#define` statements.

Most microcomputer implementations conserve precious memory space by converting the text into *tokens*, nebulous internal representations of the text. The compiler, not the programmer, cares about the form and contents of the tokens.

The committee has taken the opinion that the compiler's preprocessor must maintain the lexigraphical (spelling) information in the preprocessor, thus allowing stringizing.

Stringizing is the use of parameters with macro substitutions (`#defines`). The current definition of C allows parameters with any objects other than a string. For example, a macro definition of `sq()` can be

```
#define sq(x) x * x
```

The line

```
sq(b);
```

expands to

```
b * b;
```

Under current usage the line

```
#define PR(k, j) "printf(\"%k\\n\", j)"
```

is illegal because a macro cannot perform substitution in a string. Under a proposal the line

```
PR(d, g)
```

would expand to

```
printf("%d\n", g);
```

Details are not fully worked out. Some members advocate that escape characters not expand. Such a proposal will allow the previous macro definition to become

```
#define PR(k, j) "printf(\"%k\n\", j)"
```

thus eliminating the need to use a double backslash (\\) to form the newline escape sequence (\n).

This proposal does not negate the current rule that there can be no macro substitutions within a string. In other words, statements such as

```
#define BIG "Large"
printf("The BIG STORY is %s. \n", BIG);
```

will result in

```
printf("The BIG STORY is %s. \n", "Large");
```

The quoted BIG does not change. This rule conforms to the current definition in K & R.

const and volatile

`const` is a modifier stating that the entity which `const` modifies is nonchanging. A `const` cannot be an lvalue (assigned to, incremented, or decremented).

The type modifier `volatile` is the opposite of `const`. `volatile` types do change.

`const` and `volatile` are actually keywords informing the compiler that it can safely optimize code when

(Continued on page 5)

In this issue

que™

Letters to the editor

Q. What is `void`, the new function type?

Jim Okton
Portland, OR

A. `void` is the new function type which declares that a function returns nothing or nothing useful. This type has been added to the C vernacular for two reasons: documentation and performance. When reading the source to a program and noting that a function has been declared as `void`, you know that the return value of the function is meaningless. The resulting program also might run faster because the function doesn't need to be passed back to the invoking function. Any speed gain would be a function of the compiler implementation.

In this issue, the program `touch` uses the type `void` function call. As the program was written under a compiler that does not recognize the keyword, a `#define` was used. The form is

```
#define void int
```

which declares that any function returning a `void` actually returns an `int` instead. With a compiler that recognizes `void`, the `#define` is removed or not used.

Most of us using `void` on non-System V compilers add this line to `stdio.h` and include this file with any function using the `void` type.

/c: The Journal for C Users, is published monthly by

Que Corporation
7999 Knue Road
Indianapolis, Indiana 46250

Address all editorial correspondence to The Editor, */c*, at the address above. All requests for special permissions should be addressed to The Publisher, */c*, at the same address.

Domestic subscriptions: 12 issues, $60.00; outside U.S.A., $80.00. Single copy price, $5.00; outside U.S.A., $7.00. All funds are U.S.A. dollars. Send subscriptions, change of address, fulfillment questions, or bulk orders to

Subscriptions
/c
P. O. Box 50507
Indianapolis, Indiana 46250

Editor-in-Chief: Chris DeVoney
Managing Editor: Paul Mangin
Editorial Director: David F. Noble, Ph.D.
Copy Editor: Virginia Noble
Circulation: Sharon Sears
Cover Design: Don Tiemeier of Listenberger Design Associates

Another new data type added to C is `enum`, and a proposed data modifier is `const`. Both were briefly mentioned in the last issue. The February issue will feature further discussion on `const` and `enum`.

Q. Which function is preferable to use when you are dynamically allocating memory space: `alloc()`, `calloc()`, or `malloc()`?

Arthur Bond
Princeton, NJ

A. Throw out `alloc()`. It is an obsolete function that exists in K & R and has been kept for reasons of compatibility, but is dropped in newer standard libraries.

`malloc()` is the choice when you can exactly predetermine the size of the memory block you wish to allocate. The call for `malloc()` is

```
char *malloc(nbytes)
unsigned int nbytes;          /* number of bytes to allocate */
```

For `calloc()` the call is

```
char *calloc(n_members, member_size)
unsigned int n_members;       /* number of units to alloc */
int member_size;              /* size of each unit */
```

The `member_size` is normally performed with a `sizeof`.

`calloc()` clears the memory to zeros; `malloc()` does not.

The preference of `calloc()` over `malloc()` is principally for arrays, particularly of structures and unions. `calloc()` performs the size calculation for you. You will need to perform a separate calculation to find the size to allocate if you use `malloc()`.

Q. What's the difference between a declaration and a definition? I see both terms used interchangeably.

Tad Giming
Clearwater, FL

A. There is a small but very significant difference. A *declaration* is a statement that tells the compiler what something is (states the entity's composition) but does not allocate any memory space. A *definition* declares a data type *and* allocates space for the entity.

The following are definitions:

```
int i;

char buf[512];

static double ar[10];

struct name {
  unsigned int rec;
  char first[10];
  char last[15];
} emp;

char *change()
{
  /* code would go here */
}
```

In each case the data type is declared, and storage is allocated to hold each item, including space for the function `change()`. When a function is written, storage is allocated for the function.

The following are declarations:

```
extern int i;

char buf[];

static double ar[];

struct name {
  unsigned int rec;
  char first[10];
  char last[15];
};

char *change();
```

Notice that each of these items differs slightly from its previous counterpart.

`extern int i;` tells the compiler that you will use an `int` called `i`, but `i` has been given storage in a different module of the program. The purpose of `extern` is to inform the compiler that storage for an item has been allocated in a different module. In other words, here's the declaration, but don't allocate storage because that's been done somewhere else.

Would you believe that the second and third examples are equivalent to

```
char *buf;
static double *ar;
```

Technically, these are definitions that allocate space for pointers to the objects (array of `char` and array of `static double`) but do not allocate space for the objects that pointers point to.

The fourth example states the composition of `struct name` but does not allocate any memory to hold the structures. At this point we have declared a *structure tag* (`struct name`), not a structure. Put an identifier after the declaration, as in the first list of definitions, and you have allocated space.

The final example shows the standard declaration of a function. The key here is the semicolon that follows a declaration. No semicolon follows a definition of a function.

Q. Is there a way to access a specific memory location, such as with BASIC's PEEK statement? I need to access the file control block on a CP/M system at location 80 hex.

Mark Dilingham
Detroit, MI

A. The answer is easier than you think. Define a type `char` pointer and assign it or initalize it with 80 hex. The lines can be either

```
char *fcbp;

fcbp = (char *)0x80;
```

or

```
char *fcbp = 0x80;
```

Now you have a pointer to an array of `char`s with `fcbp` pointing to 80 hex.

The difference between the assignment (first example) and the initialization (second example) occurs because C will automatically perform the proper conversions on initializers. If you wish to play it safe, however, change the second example to

```
char *fcbp = (char *)0x80;
```

This technique of assigning or initializing a pointer for reading specific locations in memory can be used in an 8086 environment *provided* that the data segment (DS) register points to the correct segment. Such pointing is not guaranteed. For example, DOS's data transfer area (DOS's FCB) is held in the segment pointed to by the code segment (CS) register. The DS register does not always point to the same segment as the CS register.

Q. What is the advantage of using `static` in a definition?

Jack Olinski
Rhinelander, WI

A. `static` is a storage class. If a `static` variable is in a function, the variable is just like any other automatic variable *except* the `static` variable continues to exist when you leave the function. This means that the `static` variable maintains its contents. When you reenter the file, the previous value of the variable will be used.

The advantage of an internal `static` object is that it will hold its value and not disappear when the function ends.

`static` variables are initialized only once. In this declaration,

```
static int ic = 2;
```

the first time the statement is encountered, `ic` has a value of 2. If `ic` is modified within the function and the function is used again later, `ic` will hold whatever its last value was and will *not* be reinitialized to 2.

When `static` is used outside a function (an external definition), any code beyond the definition knows and has access to the entity, but the entity is unknown to any other file.

`static` gives privacy to an object. Within a function, a `static` variable is like an automatic variable and is known only to that function. When used outside a function or in a function definition, the object (a variable or a function) is known only to the functions in the file in which the object exists.

The advantage of external `static`s is that functions and data types cannot conflict with identically named functions and data types in other files. Hence, in each of four files, an external variable (defined outside a function) named `a` and a function named `b` could be declared `static`. Each variable could be different, and

(Continued on page 15)

C news

Compilers

- Manx releases new compiler versions

Manx Software Systems, the Aztec compiler company, has released Version 2 of its PC 8086 compilers and a new compiler for Apple's Macintosh computer.

Version 2.2 of the 8086 compiler produces significantly small code and faster executing programs. Another major improvement is the support of register variables. The version, which runs on the IBM PC, supports both PC DOS (all versions) and CP/M-86.

The Macintosh compiler, currently V1.06a, runs on a 128K or 512K Mac. The compiler includes a UNIX-like development shell (a subset of the Bourne shell) with a visual editor, assembler, and linker. Developed programs can run on a Mac of either size. To be added later as enhancements are a resource editor and a debugger.

The 8086 and Macintosh compiler sell for $500 each.

Manx also sells Apple, CP/M, and Radio Shack Models III and IV compilers; an Apple DOS-, PC-, PDP-11-, or VAX-to-Commodore 64 cross compiler; and a PC-, Mac-, CP/M-, or PDP-11-to-Apple cross compiler.

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07701
201/780-4004

- C compiler for HP 3000

Tymlabs adds the Hewlett-Packard HP 3000 minicomputer to the family of computers using the C language. The C/3000 compiler entered beta testing in October and should be available commercially in the first quarter of 1985.

A limited number of $1,500 discounts off the $9,000 price are given to those sites willing to use and help perfect the beta test version. The beta sites will receive the final version as well as standard warranty and support on the final version at no additional charge.

Tymlabs Corporation
211 East 7th Street
Austin, TX 78701
512/478-0611

- C compiler for Honeywell

Honeywell has announced a C compiler for its micro 6/20 and DPS-6 series computers. The compiler meets the specification as published by Bell Laboratories, and programs written under UNIX can be ported and recompiled on the Honeywell computers.

The compiler requires release 3.1 of GCOS (General Comprehensive Operating System) 6 MOD 400.

Pricing on the compiler ranges from $1,375 for micro 6/20 and DPS 6/20 computers to $2,500 for the DPS 6-45 and higher-series computers.

Honeywell, Inc.
200 Smith Street
Waltham, MA
617/895-6000

Tools

- EMACS for Rainbow 100+

UniPress Software announced the availability of the Gosling EMACS screen editor for the DEC Rainbow 100+ series. The Rainbow editor supplements versions for the IBM PC, the TI PC, and generic MS-DOS.

The Gosling EMACS is a full-function, multiple-window, MLISP (macro language) driven editor with keybindings and programming aids, such as auto-indent and parenthesis checking.

The version is priced at $375 for binary code. Source code is available on a quoted basis.

UniPress Software, Inc.
Suite 312
2025 Lincoln Highway
Edison, NJ 08817
201/985-8000

- Library for dBASE

Lattice, Inc., has announced the release of The dBC Library, a software tool kit that allows C programs to access dBASE II- and dBASE III-generated files running under MS-/PC DOS, CP/M-86, and UNIX environments.

The library can be used singularly as an indexed file system or with Ashton-Tate's dBASE products. Twenty-eight library functions provide, create, access, and update capabilities, and extend data bases. Up to eight data files and eight index files can be opened and processed simultaneously.

The library can be used with any existing dBASE application or for writing a complete application without the need for a copy of dBASE.

The dBC Library costs $250 in object form, $500 for source form. No object license is required for resold products developed with the dBC library.

Lattice, Inc.
P.O. Box 3072
Glen Ellyn, Illinois 60138
312/858-7950

Registration Cards

If you buy a book on the C language by Que Corporation, check near the back of the book for a registration card. If you find one, fill the card out and send it in. This will keep you up-to-date on any new developments on the book and you will automatically receive information on corrections to the book. It's just part of Que's commitment to our customers.

C standards

(Continued from page 1)

`const` is used but that the compiler should make no assumptions about `volatile` types.

Types

A new type, `long double`, is to be added. `long double` will support all values of `double`. This new type is machine and implementation specific regarding what additional values a `long double` can hold.

Basically, `long double` will follow `short int` and `long int`. A `short int` is guaranteed to be no larger than an `int`, and a `long int` is guaranteed to be no smaller than an `int`. But the specific ranges that `short` and `long` `int`s can use are CPU and implementation specific. On some machines a `short` integer is the same as an integer, and a `long` integer is twice the size of an `int`. On other computers an `int` and a `long int` are the same size, and a `short int` is half the size of an `int`.

Constants

With the addition of `unsigned` applying to any `int` (including `short` or `long`) and to `char`, a new suffix for constants is to be added. The suffix is `u`, which signifies an `unsigned` integer quantity. 48042u is an `unsigned` integer constant with a value of 48,042. In hexadecimal, the same quantity would be `0xbbaau`.

`u` may be used with the `l` constant modifier (`long`) to designate an `unsigned long int`.

`l` has also been added as a modifier for `long double` constants.

Either upper- or lowercase letters may be used, and the order of the letters (for `unsigned long` integers) is irrelevant. But the cases of the letters should not be mixed when an `unsigned long int` constant is used. (That is, `ul`, `UL`, `LU`, and `lu` are okay; `Ul` or `lU` may not be acceptable.)

Case

One of the possible changes may be made in the `case` statement. A proposal allows a range of constant expressions in a single `case` statement. The operator for this range is not yet known.

The result of this change will allow

```
case '0':
case '1':
case '2':
case '3':
case '4':
```

to become

```
case '0' .. '4':
```

where .. is the tentative range operator. Be aware that the ability to use a range does not negate the need for multiple case statements. The statement

```
case 'P' .. 'p':
```

is not the same as

```
case 'P':
case 'p':
```

The first example gives all constant integer values between P and p inclusively. The second example has only the constant integer values P and p.

A note

Please remember that all information presented here is tentative, and the proposed standard is just that—*proposed*—not final or binding. This information, based on current discussions within the committee, is subject to change and refinement. Do not expect any C compiler to incorporate these changes. Until the draft becomes a final adapted ANSI standard (in approximately December, 1986), compiler publishers are not bound to follow these guidelines.

If you have any questions or comments about the draft that is evolving toward a C standard, write to us at */c*. We will be happy to pass responses and questions to the committee and answer questions in this publication.

Functions and articles wanted

/c is looking for authors of functions and articles on the following subjects:

Tips on the C language
C Programming techniques
UNIX
Application and programming tools
Discussion and explanations of source code

The tone and content should be "moderately" technical. If any code is specific to an environment or compiler, the host environment or compiler should be clearly identified.

Writers should include a brief outline; clips of current works (if any); a current resume (if available); a telephone number; and a self-addressed, stamped envelope. We reply to submissions as soon as possible. Submissions used are paid on publication of the article or item.

Submit double-spaced typewritten copy with standard margins, or submit single-spaced text on CP/M (8-inch) or IBM PC-compatible (5 1/4-inch) diskettes.

Accompanying photos, sketches, diagrams, graphs, and charts are encouraged.

Length of articles: 300-1,000 words for news, 1,000-5,000 words for features, but no set length for source code.

Submissions and inquiries should be sent to

/c Editor
c/o Que Corporation
7999 Knue Road
Indianapolis, IN 46250

All material accepted for publication in */c* becomes property of Que Corporation.

touch: a program to set a file's date and time

The program touch is an MS-DOS port of the UNIX program. touch maintains the flavor of the UNIX version.

In UNIX, a file has both creation date-and-time stamp, an access date-and-time stamp, and a last modification date-and-time stamp. MS-DOS maintains only one date-and-time stamp. The UNIX touch updates the access and/or modification stamp. This touch will update only the single date-and-time entry in the disk directory.

The program presents several challenges, which include writing a program in C to perform operating system calls and maintaining the style of the UNIX version of touch.

As the program is written for MS-DOS, which runs only on 8086-family computers, no attempt has been made to make the program portable to other processors. An attempt, however, has been made to let the program and its underlying function port between 8086 compilers running under MS-DOS. For this reason, two versions of some accompanying support functions and modules have been presented, one written under Computer Innovations C86 Version 2.x, and one under Lattice C for the 8086, Version 2.1x.

Description

Usage from the command line (only) is

```
touch [MMDDhhmm[YY]] filename filename ...
```

where

MM is the two-digit month
DD is the two-digit day
hh is the two-digit, twenty-four-hour time
mm is for two-digit minutes
YY is the two-digit year (1900 is assumed)

all dates/times are optional

filename is the name of the file to be updated
drive and path names are allowed
multiple file names are allowed
ambiguous file names are *not* allowed

If no date or time is given, the current system date and time are used. If only a part of the date or time is given, the remaining parts are taken from system date and time.

The date and/or time must be given in order, but any part following the specified date and/or time is optional. To set the month, give only the two-digit month. To set the day, give the month and day. To set the minutes, give the month, day, hour, and minutes.

Diagnostics

If the file's date and time are successfully set, no message is displayed. If a file is not found (because the incorrect drive, path, or file name was given; the file does not exist; or an ambiguous file name was used), an error message is displayed on stderr, and the file is skipped.

If an improper date or time is given, DOS refuses the function call, causing touch to fail on all given file names.

The exit level of the program is set to the number of files whose dates and/or times were not updated. An exit level of zero indicates that all files were updated.

Compiling and linking

To compile the modules, you will need the header files, stdio.h, segstr.h, and dtstr.h.

If you are using the Lattice compiler, change the value of C86 to zero in the modules touch2.c and getdt.c. Compile the modules touch1.c, touch2.c, and getdt.c. If you are using Computer Innovations' C86 compiler, also compile getds.c.

The link order is touch1, followed by touch2 and getdt. For C86 users, also link getds.

For MS-DOS LINK and CI-C86 users, the link command line is

```
link touch1 touch2 getdt getds, touch, , c86s2s
```

The link command line for Lattice's small model is

```
link \s\c touch1 touch2 getdt, touch, , s\lc
```

To test the function, create a dummy disk file (or files) and use touch. We highly recommend that you do not use touch on live files during development until you have satisfied yourself that the program is working correctly.

Construction of touch

The core of the touch program is divided into two modules. The first module (touch1.c) is compiler-independent. The second module (touch2.c) isolates compiler dependencies into this single module. The difference between the two example compilers, Computer Innovations and Lattice, is in their function that invokes DOS function calls.

CI uses sysint21(), and Lattice uses intdos(). Using a different function for each compiler is one change that must be made when porting the program between these two compilers.

The second change is that the structure for holding the information for the 8086 registers is dissimilar. CI allows the setting of the data segment (DS) register in its sysint21() call. Lattice does not allow setting the DS register in intdos(). In a small memory module program (a data segment less than 64K), the DS register must read and be set for CI's sysint21() function. For Lattice and small module programs, the DS register can be ignored.

For this reason, a #define C86 is used near the start of touch2.c (and getdt.c). If C86 is nonzero, the appropri-

ate `#if C86` preprocessor directives will compile the code for C86. A zero value for `C86` causes the compiler to invoke the appropriate DOS calling function for Lattice.

With this background, we'll highlight the flow of the program.

The first few lines after the opening comments call for the proper header files to be included. `segstr.h` is the header file for the DOS function call function. `dtstr.h` holds the date and time structures used for getting the DOS system date and time.

Following the includes, macros in `touch2.c` are defined for the value of some DOS function calls. Macros are defined in `touch1.c` for signaling either proper (GOOD) or improper (ERROR) completion of a function.

In `touch1.c` two global `unsigned ints` are defined that will hold a packed system date and time returned from DOS. Another integer, `badfiles`, is defined that will hold the number of files which cannot be touched.

`main()` is defined with the proper `argc` and `argv` so that arguments from the command line can be processed. After some definitions and declarations, `main()` begins its processing.

Getting the date and time of the files is the next noteworthy feature. The test focuses on `argv[1]`. If the person has given a date or time argument, `argv[1]` must be nonzero. (The month, which is always the first part of the date/time argument, can never be zero.) If `argv[1]` is nonzero, we gather the argument, call `mkptd()`, and bump `argc` and `argv`. The bump, the decrementing of `argc` and incrementing of `argv`, ensures that the date and time are not processed as a file name.

If `argv[1]`'s integer value is zero, the `argv[1]` is a file name to be processed. Because no date and time are given, the current system date and time are gathered and converted into a file date and time.

The `while` loop is used to pass arguments to the function `touch()` until all arguments have been processed.

Other than `main()`, all functions in a module are presented in alphabetical order. This arrangement conforms to the prevalent, but not absolute, UNIX practice of organizing functions in a module.

`mkptd()` (make a packed time and date) accepts a character string and makes the packed file date and file time wanted by DOS. The function uses the date and time structures from `dtstr.h`.

The function first passes the date and time structures to `_getdate()` and `_gettime()`, respectively. These functions, shown in the listing of `getdt.c` (get system date and time), gather the system date and time into the structures.

The function then calls `sscanf()`. The conversion argument calls for a maximum of five two-digit integers to be drawn from the string passed to `mkptd()`. Because the arguments passed to `sscanf()` must be pointers, the address-of operator (&) is used to pass the address of each member of the time and date structures.

Passing simply the members of the structures without the address-of operator would be disastrous. `sscanf()` would place the resulting values from the string in memory (s) into the memory locations pointed to by the value of the structure members' contents, not by the address of the structure members. This action could play havoc with our program, operating system, or computer because `sscanf()` places the formatted input into memory locations based on the system date and time. `sscanf()` requires pointers as arguments, and the address-of operator is thus needed.

`sscanf()` has an interesting feature. If enough input is not found in the string in memory to satisfy all conversions, any unfulfilled argument is left untouched. In other words, if only a month and a day are entered, only the first two `%2d` conversions are fulfilled. The remaining three conversions for the hour, minute, and year are unfulfilled, and the contents of these three remain unchanged.

This same feature does not apply to `scanf()` and `fscanf()`. Both functions will continue to absorb input (either from the console or the stream, respectively) until the conversion is complete. The difference is in the handling of EOF. All three scan functions end when EOF is encountered. When information is obtained from a string in memory, a null is equvalent to EOF. To terminate input to `scanf()` or `fscanf()`, an EOF character must be entered or encountered.

This feature of `sscanf()`'s allows us to fill the structure date and time, call `sscanf()`, and know that only the given arguments on the command line for the date and time will be set. The remaining members will keep their values from the earlier call for the system date and time.

Moving to the `touch()` function, you'll note two nonstandard functions: `sopen()` and `sclose()`. These functions are defined in `touch2.c` Why use these functions when `open()` and `close()` are available?

The answer is that the DOS function to set a file's date and time needs a file handle. More specifically, the DOS function needs the file handle passed back to the program from the DOS open file call.

On MS-DOS computers, five files are automatically opened. The three typical UNIX-like files are stdin (fd = 0), stdout (fd = 1), and stderr (fd = 2). However, also opened are the list device, prn (fd = 3), and the communications port, aux (fd = 4).

What happens when your compiler opens its first file? The first file handle number that DOS will give to the compiler is 5. Most C compilers, however, attempt to follow the UNIX pattern of only three default open files. Therefore, the compiler will generally translate the file

handle returned by DOS into a different numbered `fd` that is returned from `open()`. This number is typically 3. Because of this internal translation by the compiler, the DOS file handle and the compiler's `fd` are not the same.

Earlier versions of this program failed. The reason is that the DOS call to change the file date and time used the file descriptor returned by the compiler's `open()`, not the handle returned by DOS. DOS simply accepted the call and did nothing. An error was not returned.

For this reason `sopen()` and `sclose()` have been written to obtain a file handle from DOS directly so that the file's date and time can be sucessfully set. Both functions call DOS by using the compiler's DOS-calling function [`sysint21()` or `intdos()`].

`chftd()`, also in `touch2.c`, simply calls DOS with the function call number for setting a file's date and time and the handle number. You can see how the 8086's registers are loaded to accommodate the call.

Some manipulations occur in the functions `mfdate()` (make file date) and `mftime()` (make file time). These manipulations occur because of the way DOS stores the date or the time in a single 16-bit word.

The format for a file date and time is shown in figure 1. Both `mfdate()` and `mftime()` use left-shifts and ors to put the components of the date and time into their proper places. Declaring `pdate` and `ptime` as `unsigned ints` guarantees that the bits are zero filled on the shift. Zero-filled bits are not important in this program because no integer is shifted over a 16-bit boundary. However, shifting `signed` integers does not cause zero fills, and the difference between shifting signed and unsigned integers can be an issue in other programs.

`touch2.c` and `getdt.c` show methods to tackle an interesting problem. How do we cope with using DOS function calls when we can make few assumptions about the physical makeup of our variables? When calling DOS, we must load the 8086's registers. How can this be accomplished?

Examine the file `segstr.h`. The structures in this file are used in the DOS call. The first structure declares a series of `unsigned ints`, with each `unsigned` integer representing the various full registers to be loaded (the AX, BX, DX, etc.). The next structure is based on the knowledge of the 8086 and all 8086 C compilers. An integer uses two bytes: a low byte followed by a high byte. The registers of the 8086 follow this convention. To access half registers, such as AH and AL, we declare the structure to hold `unsigned chars`, which are eight bits in length. As Lattice does not accept `unsigned char`, we simply use `char` instead. Thus, we can construct two structures and one union, any of which may be used to load the DOS call function.

The use of either of these structures requires that an integer and full register are 16 bits wide, a character and half register are 8 bits wide, and integers are stored low-high. Therefore, the program will work perfectly with any 8086-based C compiler but will fail if moved to a different processor, such as an MC68000.

When you work with a given processor and environment (MS-DOS), some C code will simply be nonportable. Very little way exists to construct a portable version of this program. The code ports to other compilers in the MS-DOS and 8086 realm but does not port to any other environment without modification.

The union in `segstr.h` is simply a convenient way to load any full or half register of the 8086. An example is evident in `getdate()`, where the AH register receives the function call number; the DH and DL half registers get the month and day, respectively; and the CX gets the year. The same effect can be achieved by shifting and "oring" the AX and DH registers. This method of using the union makes the code more understandable, but the price for this legibility is increased code size. The compiler needs to store and manipulate the offsets into the structures and union. Thus, the size of the code will generally increase.

Summary

`touch` is a program that demonstrates an approach to several problems. The problems of using operating system calls, loading CPU registers, emulating UNIX-like features, and generalizing support functions are tackled in this program and accompanying functions. `touch` simply presents a way to overcome these obstacles. Obvious improvements may be made.

Figure 1

DOS packed file date

< most signficant byte >								< least signficant byte >							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
y	y	y	y	y	y	y	m	m	m	m	d	d	d	d	d
|	year						|	month			|	day		|

where

bits 15-9 are the binary year (1980-2099)
bits 8-5 are the binary month (1-12)
bits 4-0 are the binary day (1-31)

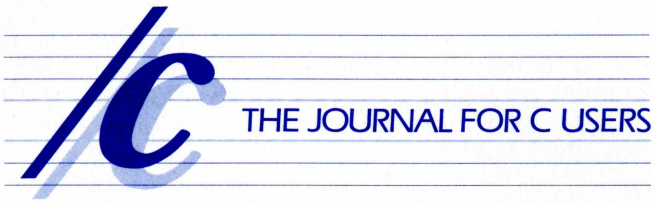
DOS packed file time

< most signficant byte >								< least signficant byte >							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
h	h	h	h	h	m	m	m	m	m	m	i	i	i	i	i
|	hours		|	minutes			|	two-sec increments						|

where

bits 15-11 are the binary hours (0-23)
bits 10-6 are the binary minutes (0-59)
bits 4-0 are the number of two-second increments

SUBSCRIBE TO . . .



You don't want to be without valuable, up-to-date information on the C language. Each monthly issue provides:

- Valuable C programming techniques
- Clear, concise explanations of functions
- Exciting tips on the C language
- And more . . .

A "moderately technical" journal, ***/C*** is sophisticated enough for the experienced C programmers, yet within reach of newcomers.

This convenient, postage-free card makes subscribing easy. Just fill in the order form below and mail it with your payment. Or check the box marked "Bill Me," and your payment won't be due until after you receive your first issue. For faster service, call our toll-free order line (**1-800-428-5331**) and use your credit card information TODAY. Take your pick of payment options, but subscribe NOW!

CALL NOW OR USE THIS CONVENIENT FORM TODAY.

(CUT HERE)

YES, I want */C.*

☐ Send me 12 monthly issues for $60 (outside U.S.A., $80)*

☐ Send me 24 monthly issues for $110 (outside U.S.A., $145)*

*All money must be payable in U.S. funds.

S

Company Name (if applicable) _____

Attention of: (Name) _____

Address _____

City _____ State _____ ZIP _____

Method of Payment:

☐ Check ☐ VISA ☐ MasterCard ☐ AMEX ☐ Bill Me

Cardholder Name _____

Card Number _____ Exp. Date _____

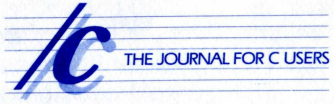
Signature _____

(for credit card)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Call Permit No. 9918 Indianapolis, IN

Postage will be paid by addressee



P.O. Box 50507
Indianapolis, IN 46250

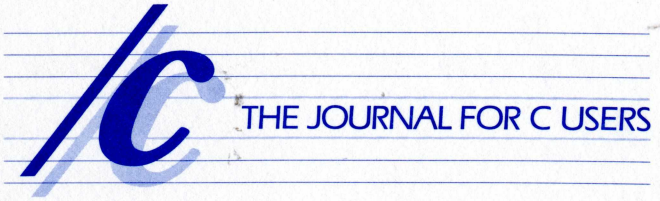
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Call Permit No. 9918 Indianapolis, IN

Postage will be paid by addressee

P.O. Box 50507
Indianapolis, IN 46250

SUBSCRIBE TO . . .



You don't want to be without valuable, up-to-date information on the C language. Each monthly issue provides:

- Valuable C programming techniques
- Clear, concise explanations of functions
- Exciting tips on the C language
- And more . . .

A "moderately technical" journal, ***/C*** is sophisticated enough for the experienced C programmers, yet within reach of newcomers.

This convenient, postage-free card makes subscribing easy. Just fill in the order form below and mail it with your payment. Or check the box marked "Bill Me," and your payment won't be due until after you receive your first issue. For faster service, call our toll-free order line (**1-800-428-5331**) and use your credit card information TODAY. Take your pick of payment options, but subscribe NOW!

CALL NOW OR USE THIS CONVENIENT FORM TODAY.

(CUT HERE)

YES, I want */C.*

☐ Send me 12 monthly issues for $60 (outside U.S.A., $80)*

☐ Send me 24 monthly issues for $110 (outside U.S.A., $145)*

*All money must be payable in U.S. funds.

Company Name (if applicable) _____

Attention of: (Name) _____

Address _____

City _____ State _____ ZIP _____

Method of Payment:

☐ Check ☐ VISA ☐ MasterCard ☐ AMEX ☐ Bill Me

Cardholder Name _____

Card Number _____ Exp. Date _____

Signature _____

(for credit card)

segstr.h for C86 (tabs = 3)

```
/*    structures for sysint() and segread()  */
/*                                           */
/*    CI-C86 compiler  PC/MS-DOS             */
/*                                           */
/*    sysint() and sysint21() structs/union  */

struct xregs {              /* full registers */
   unsigned int ax,
                bx,
                cx,
                dx,
                si,
                di,
                ds,
                es;
};

struct hregs {              /* half registers */
                            /* (low-high)     */
   unsigned char  al, ah,
                  bl, bh,
                  cl, ch,
                  dl, dh,
                  sl, sh,
                  dil, dih,
                  dsl, dsh,
                  el, eh;
};

union REGS {      /* access regs either way */
   struct xregs XREGS;
   struct hregs HREGS;
};

/* segread() structure  */

struct segs {
   unsigned int scs,
                sss,
                sds,
                ses;
};
```

dtstr.h (tabs = 3)

```
/*    date & time structures     */

struct date {
   unsigned int month;
   unsigned int day;
   unsigned int year;
};

struct time {
   unsigned int hour;
   unsigned int minute;
   unsigned int second;
   unsigned int thousand;
};
```

segstr.h for Lattice (tabs = 3)

```
/*    structures for intdos() and segread()  */
/*                                           */
/*    Lattice 8086   PC/MS-DOS               */
/*                                           */
/*    intdos() and intdosx() structs/union   */

struct xregs {              /* full registers */
   unsigned int ax,
                bx,
                cx,
                dx,
                si,
                di;
};

struct hregs {              /* half registers */
                            /* (low - high )  */
            char  al, ah,
                  bl, bh,
                  cl, ch,
                  dl, dh,
                  sl, sh,
                  dil, dih;
};

union REGS {      /* access regs either way  */
   struct xregs XREGS;
   struct hregs HREGS;
};


/* segread() structure  */

struct segs {
   unsigned int es,
                cs,
                ss,
                ds;
};
```

(Continued on next page)

Quickie

Is the following program legal? If not, why not? If so, what does it produce?

```
#include "stdio.h"
main()
{

    int i;

    for(i = 0; i < 16; ++i)
        putchar("0123456789abcdef"[i]);

}
```

(Answer on page 15)

touch1.c file (tabs = 5)

```
/*        touch (update file's date and time) program                    */
/*                                                                       */
/*        Written for CI-C86 V2.x & Lattice C V2.1x or later             */
/*        MS-/PC DOS V2.x or later                                       */

#include "stdio.h"
#include "dtstr.h"        /* date and time structure     */

#define ERROR    -1       /* error signal                               */
#define GOOD     0        /* okay signal                                */

unsigned int pdate,       /* packed date for DOS */
             ptime;       /* packed time for DOS */

int badfiles = 0;         /* number of files not updated */

main(argc, argv)
int argc;
char **argv;
{

     struct date cdate;          /* struct for current date */
     struct time ctime;          /* struct for current time */
     unsigned int mfdate();      /* declare mfdate() and mdtime() to  */
     unsigned int mftime();      /* return unsigned ints     */

     if(argc < 2) {
          puts("usage: touch [mmddhhmm[yy]] filename filename ...");
          exit(GOOD);
     }
     if(atoi(argv[1])) {
          mkptd(argv[1]);
          ++argv;
          --argc;
     } else {
          _getdate(&cdate);              /* get the current date  */
          pdate = mfdate(&cdate);        /* set pdate to packed file date */
          _gettime(&ctime);              /* get the current time  */
          ptime = mftime(&ctime);        /* set ptime to packed file time */
     }
     while(--argc > 0)
          badfiles += touch(*(++argv));
     exit(badfiles);

}

unsigned int mfdate(p)          /* make a packed file date from a given date */
struct date *p;                 /* struct that holds the date           */
{

     unsigned int pdate;        /* to hold packed date         */

     pdate = (p->year - 1980) << 9;     /* left shift year to bit 9   */
     pdate |= p->month << 5;            /* or in month at bit 5       */
     pdate |= p->day;                   /* or in day                  */
     return(pdate);                     /* return the packed date     */

}
```

```
unsigned int mftime(p)          /* make a packed file time from a given time */
struct time *p;
{

     unsigned int ptime;        /* to hold packed time    */

     ptime = p->hour << 11;          /* put the hour in bit 11 */
     ptime |= p->minute << 5;        /* or in mintue at bit 5  */
     ptime |= (p->second / 2);       /* or in two-second increments at bit 0 */
     return(ptime);                  /* return the packed time */

}

int mkptd(s)                    /* make pdate and ptime from s */
char *s;
{

     struct date tdate;         /* struct to hold today's date */
     struct time ttime;         /* struct to hold today's time */
     unsigned int mfdate();     /* declare these return unsigned ints */
     unsigned int mftime();

     _getdate(&tdate);          /* get the current date */
     _gettime(&ttime);          /* and the current time  */
     /* put altered date/times into structures */
     sscanf(s, "%2d%2d%2d%2%2d", &tdate.month, &tdate.day,
          &ttime.hour, &ttime.minute, &tdate.year);

     pdate = mfdate(&tdate);
     ptime = mftime(&ttime);
     return;

}

int touch(s)               /* start of touch function */
char *s;                   /* file name to touch      */
{


     unsigned int handle;          /* file handle returned from sopen() */
     unsigned int sopen();         /* declare sopen() returns unsigned */

     if((handle = sopen(s)) == ERROR) {          /* file didn't exist */

          printf("touch: I cannot find %s!\n", s);
          return(1);
     }
     if((chfdt(handle, pdate, ptime)) == ERROR) { /* something went wrong */

          printf("touch: I cannot update %s\'s date and time!\n", s);
          sclose(handle);
          return(1);
     }
     sclose(handle);
     return(0);

}
```

touch2.c file (tabs = 5)

```
/*        touch2 - compiler specific support functions                */
/*                                                                    */
/*        Written for CI-C86 V2.x & Lattice C V2.1x or later          */
/*        MS-/PC DOS V2.x or later                                    */

#include "stdio.h"
#include "segstr.h"       /* struct for DOS calls        */
#include "dtstr.h"        /* date and time structure     */

/*   If using C86, set C86 to 1          */
/*   If using Lattice, set C86 to 0      */

#define C86          1

#define OPENFFN  0x3d00  /* open file fn call                        */
#define CLOSFFN  0x3e00  /* close a file handle fn call              */
#define GETFDT   0x5700  /* get/set file date/time fn                */
#define SETFDT   0x5701  /* get/set file date/time fn with set       */
#define ERROR    -1      /* error signal                             */
#define GOOD     0       /* okay signal                              */


int chfdt(fh, fdate, ftime)   /* change a file's date & time  */
unsigned int fh;              /* file handle to update */
unsigned int fdate;           /* new file date         */
unsigned int ftime;           /* new file time         */
{

     struct xregs reg;              /* DOS call structure    */
     register struct xregs *r;

     r = &reg;                      /* point r to reg        */

     r->ax = SETFDT;                /* set file date/time fn -> ax     */
     r->bx = fh;                    /* file handle -> bx     */
     r->dx = fdate;                 /* file date -> dx       */
     r->cx = ftime;                 /* file time -> cx       */

#if C86
     if(sysint21(r, r) & 1)         /* DOS rejected call     */
#else
     if(intdos(r, r) & 1)           /* DOS rejected call     */
#endif
          return(ERROR);
     else
          return(GOOD);

}

int sclose(fh)           /* special close for file handle   */
unsigned int fh;         /* file handle to close       */
{

     struct xregs reg;
     register struct xregs *r;

     r = &reg;                  /* point r to reg        */

     r->ax = CLOSFFN;           /* close file fn -> ax   */
     r->bx = fh;                /* file handle -> bx     */
```

```
#if C86
     if(sysint21(r, r) & 1)
#else
     if(intdos(r, r) & 1)
#endif
          return(ERROR);       /* carry flag showed error    */
     else
          return(GOOD);

}

unsigned int sopen(s)     /* special open to get file handle */
char *s;                  /* name of file to open       */
{

     struct xregs sreg, rreg;              /* DOS call structs      */
     register struct xregs *sp, *rp;       /* pointers for structs  */
#if C86
     unsigned int getds();              /* get data segment function  */
#endif

     sp = &sreg;                    /* point s to sreg and   */
     rp = &rreg;                    /* r to rreg             */

     sp->ax = OPENFFN;              /* open file fn -> ah    */
                                    /* access code (Ø is r/o) -> al */
#if C86
     sp->ds = getds();              /* ensure ds is correct  */
#endif

     sp->dx = (unsigned int)s;      /* pointer to s in dx    */

#if C86
     if(sysint21(sp, rp) & 1)       /* error occurred        */
#else
     if(intdos(sp, rp) & 1)         /* error occurred        */
#endif
          return(ERROR);            /* return bad            */
     else
          return(rp->ax);           /* else return handle    */

}
```

getds.c file for C86 (tabs = 3)

```
/*    get data segment for small model    */
/*                                        */
/*    CI-C86                              */

#include "segstr.h"     /* segread() and sysint() structure */

unsigned int getds()    /* get data segment */
{

   struct segs seg;

   segread(&seg);     /* do the segread */
   return(seg.sds);   /* return the data segment as unsigned int */

}
```

getdt.c file (tabs = 5)

```
/*   get date/time functions for MS-DOS      */
/*                                           */
/*   CI-C86 V2.x and Lattice V2.1x or later  */
/*   MS-/PC DOS V2.x or later                */

#include "stdio.h"
#include "segstr.h"
#include "dtstr.h"

/*   If using C86, set C86 to 1          */
/*   If using Lattice, set C86 to 0      */

#define C86    1

#define GDATEFN 0x2a                /* get date function call # */
#define GTIMEFN 0x2c                /* get time function call # */

void _getdate(p)                    /* get system date  */
register struct date *p;            /* pointer to struct to receive date */
{

     union REGS reg;
     register union REGS *r;

     r = &reg;                      /* point r to regs   */

     r->HREGS.ah = GDATEFN;         /* get date fn -> ah */

#if C86
     sysint21(r, r);                /* go system call */
#else
     intdos(r, r);                  /* go system call */
#endif

     p->month = r->HREGS.dh;        /* file date struct */
     p->day = r->HREGS.dl;
     p->year = r->XREGS.cx;
     return;

}

void _gettime(p)                    /* get system time */
register struct time *p;            /* pointer to struct to receive time */
{

     struct hregs reg;
     register struct hregs *r;

     r = &reg;                      /* point to reg struct */
     r->ah = GTIMEFN;               /* get time fn -> ah */

#if C86
     sysint21(r, r);                /* do the system call */
#else
     intdos(r, r);                  /* do the system call */
#endif

     p->hour = r->ch;               /* set members of time struct */
     p->minute = r->cl;
     p->second = r->dh;
     p->thousand = r->dl;
     return;

}
```

Letters

(Continued from page 3)

each function could do entirely different things. When linked together, none of the `static a` variables and none of the `b()` functions would conflict with each other.

Q. Can comments be nested? Some compilers allow nested comments; others do not. I find nested comments useful when I need to comment out a section of code for debugging.

Bryan Molier
Hamilton, Ontario

A. Kernighan and Ritchie and the current feeling of the J11 C standards committee is that comments do *not* nest.

Nested comments are indeed useful. By adding an opening comment (`/*`) at the beginning of the code and a closing comment at the end (`/*`), you can quickly comment out a section of code.

If I can offer a suggestion, to ensure compatibility regardless of whether a compiler respects nested comments, use

```
#if 0
```

at the beginning and

```
#endif
```

at the end of the code you wish to "comment out." Because `#if 0` always evaluates to false, the section between this preprocessor directive and the corresponding `#endif` will not be compiled.

I should mention that if your compiler does not support nested `#if` statements and you have `#if`, `#ifdef`, or `#ifndef` directives in this chunk of code you wish to comment out through the `#if 0`, this technique will not work. However, most compilers respect nested `#if`s.

Corrections and Clarifications

In the November issue, the `gets()` function has an error. Change `p` to `s`. The line should read

```
if (c == EOF && s ==line)
```

In the **Letters to the Editor** column, we stated that Lattice used a switch to pool literal strings. This is true of Version 2.0x of the compiler, but not true in Version 2.1x. Version 2.1x automatically pools identical string constants; i.e., only one copy of identical constant strings are kept in memory. For the following example

```
char buf [] = "abcdefg";
char nuf [] = "abcdefg";
```

only one copy of `abcdefg` will be kept in memory.

Quickie Answer

The program is perfectly legal. The program will print `0123456789abcdef`, a poor man's hexadecimal converter.

The possible confusion lies in the line

```
putchar("0123456789abcdef"[i]);
```

`i` is an integer, and `"0123456789abcdef"` is a character array. When the array is encountered in a program, the characters are stored in the literal pool, and the array becomes an address. Hence

```
"0123456789abcdef"[i]
```

becomes "the character at the address of (the array plus i)". This is the same as having written

```
char s[] = {"123456789abcdef"};
main()
{
    . . .
    putchar(s[i]);
```

Our thanks to Jim Flemming, who pointed out this classic C example to us.

Conventions

A set of conventions is used in */c* to clarify programs and text.

All computer-generated or program code (such as . c and . h files) appears in a digital typeface `like this`. If input is given, the code is shown in an italic typeface *like this.*

In some programs tabs are condensed to every three or five columns, not the normal eight. This modification allows a more lucid presentation of heavily-indented code. If your editor does not allow the setting of tab stops or if such settings are not convenient, adjustments to trailing comments may be made if proper alignment is desired. A line above the program will indicate whether the tab stops have been altered.

The digital typeface and a ruler line are reproduced below to aid in reading programs.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()_-=+{}[]""'~`/?.,<>:;
0  0  1  1  2  2  3  3  4  4  5  5  6  6  7  7  8
1  5  0  5  0  5  0  5  0  5  0  5  0  5  0  5  0
```

C quiz

Every month /c will be offering a new quiz. For each quiz, the first correct response received at our offices at /c will get a small prize. In case of a tie, either duplicate prizes will be awarded or the earliest postmark will win. (This decision will be made at the whim of the editor!) All entries must be mailed.

Last month's quiz:

With the exception of control and alphanumeric characters, what ASCII characters have no predefined meaning in C? (The ASCII set is defined as 7-bit codes.)

The answer:

dollar sign	$	0x24	044
at sign	@	0x40	100
underscore	_	0x5f	137
grave (accent)	`	0x60	144

Of these four characters, only the underscore may appear in a name. If we want to split hairs, the sharp or pound sign (#, 0x23) is not recognized by C either. This symbol *is* recognized by the preprocessor.

The winner of the November quiz will be announced in the January issue.

December's quiz

Name the characters that have multiple uses in C. For instance, the equal sign (=) is used for assigment (=) and equality (==). Disregard other combinations that use a single equal sign, such as OR equal (|=).

Send your answers to

/c Quiz
c/o Que Corporation
7999 Knue Road
Indianapolis, IN 46250

The winner of this month's quiz, to be announced in the February issue, will receive a copy of Que's upcoming book by Kim Brand, *Common C Functions*.

Editor survey

To harry the beginning C programmer, C requires that a person learn not only the language but also a text editor. We at /c are interested in what text editor you use. Drop us a letter or postcard with the name of your favorite editor, its source or publisher, your environment (both operating system and computer model), and an optional line or two about why you use this editor. Send the information to

/c Editor Survey
Que Corporation
7999 Knue Road
Indianapolis, IN 46250

We'll compile the results of the survey and publish the findings in the March issue.

C humor: the D language

by Chris DeVoney and Jack Purdum

If you recall from the November issue, the authors proposed the D language to supersede the C language. D adds functions and features unknown to (and unwanted by) the original designers of C. Below is a list of more D functions.

```
int bgetc()

int bputc(c)
int c;
```

Backwards `getc()` and `putc()`.

```
double crash(item)
unsigned item;
```

Immediate crash of computer system component `item`. Returns cost of repair in dollars and cents.

```
char *fuzz()
```

Replaces data found with fuzz and returns character pointer to nonsense.

```
int plug_eject()
```

Ejects power cord from ac wall receptacle. Returns frustration as an `int`.

```
int rotate_45(d)
int d;
```

Slows down minifloppy drive `d` to accommodate 45 rpm records. Eight-inch drives should use `rotate_33()`.

```
char *zap_ac()
```

Returns 110 volts down the terminal line.

The authors of the D language are looking for additional operators, functions, and features. For your chance at immortality, send your suggestion to

/c D
c/o Que Corporation
7999 Knue Road
Indianapolis, IN 46250

If your suggestion is printed, you'll receive credit and five dollars. All suggestions printed become property of Que Corporation.

Coming in the January issue of /c

Coverage of the winter quarterly meeting of the ANSI J11 committee, the C standards committee.

Making your programs smaller with one easy function.

getopt, a function to get command line arguments.

Plus

Letters to the editor

Our monthly quiz

and more. . . .